Master's Thesis

NeuroValve: Energy-efficient Deep Learning Inference on Mobile Devices Exploiting Neural Network Architectures

Gyungmin Bin

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

2021

NeuroValve: Energy-efficient Deep Learning Inference in Mobile Devices Exploiting Neural Network Architectures

Gyungmin Bin

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

NeuroValve: Energy-efficient Deep Learning Inference in Mobile Devices Exploiting Neural Network Architectures

A thesis/dissertation submitted to
Ulsan National Institute of Science and Technology
in partial fulfillment of the
requirements for the degree of
Master of Science

Gyungmin Bin

2020/01/04

Approved by

Advisor
Youngbin Im

NeuroValve: Energy-efficient Deep Learning Inference in Mobile Devices Exploiting Neural Network Architectures

Gyungmin Bin

This certifies that the thesis/dissertation of Gyungmin Bin is approved.

2020/01/04

Signature

Advisor: Youngbin Im

Signature

Kyunghan Lee

Signature

Hyoil Kim

Abstract

On-device deep convolutional neural network (CNN) inference is often desirable for user experience and privacy. However, running CNN inferences on mobile devices is particularly challenging due to their processing, memory, and battery limitations. To tackle this challenge, we focus on the block-wise properties of CNNs in which several blocks with configurable parameters are sequentially and repeatedly processed. Our measurement study reveals that such block-wise workloads can be characterized well with a metric called memory access rate (MAR), which is affected by a CNN block's configuration, the cache size of the device, CPU frequency, and memory bandwidth. With this understanding of the workload from the blocks in state-of-art mobile CNNs and their impact on the energy consumption and processing delay, we propose NeuroValve, a fine-grained CPU frequency and memory bandwidth scaling framework for mobile CNN inference. Our performance evaluations with NeuroValue implemented on off-the-shelf smartphones over various state-of-the-art CNN models show that NeuroValve substantially saves energy consumption while experiencing a minor increase in the processing delay compared to the Android default governor and an advanced memory bottleneck-aware governor.

Contents

List of Figures

List of Tables

I Introduction

Deep neural networks (DNNs) have become one of the most widely used techniques in various artificial intelligence applications. Among others, convolutional neural networks (CNNs) have shown their superiority in a wide range of application domains such as image recognition [53, 25, 54, 28], object detection [46, 47, 24], mobile sensing [58], and semantic segmentation [40, 9, 2]. The execution of such CNN models typically requires high computational power due to large and complex network structures, stressing resource-constrained mobile devices. For running CNNs on mobile devices, recent approaches propose intelligent cloud-based offloading [34, 36, 60], which splits the CNN model into two segments: one segment running on the mobile device and the other on the server. Although they can reduce the amount of data to be offloaded to the server and save the mobile device's computational energy consumption compared to conventional offloading techniques, they still suffer from network disruptions and privacy concerns. As a result, despite the challenges of constrained resources, it becomes more compelling to make inferences on the mobile device for time-critical or privacy-sensitive services through various hardware/software-based techniques.

To this end, several recent works have focused on improving on-device inference in terms of energy efficiency, accuracy, and latency. For instance, model compression techniques such as quantization [32] and pruning [23, 37, 26] are known to enhance computational efficiency while maintaining neural network accuracy. Several recent works also built compact CNN architectures that balance the trade-off between the accuracy and computation complexity [30, 28, 61, 48]. Besides these, efforts to redesign hardware have also been made for more efficient neural network processing [33, 22, 42].

Despite the efforts above, it remains unexplored how to run CNN models energy efficiently given a set of software and hardware. More specifically, how to control the hardware for CNN workloads through the energy management subsystems, which exist in every mobile operating system, is still an open question. For instance, the Linux kernel on Android devices has subsystems called `cpufreq` to manage the CPU frequency and `devfreq` for other components such as memory. With these subsystems, a software module known as a *governor* implements an algorithm that controls those frequencies based on its criteria [7] to balance between performance and power consumption.

However, controlling such subsystems for CNNs via dynamic voltage and frequency scaling (DVFS) on mobile devices is challenging for two reasons. *First*, DVFS governors tailored for general purpose cannot meet the characteristics of a variety of DNNs on mobile devices, thus exhibiting poor energy efficiency for many neural network applications [45]. *Second*, existing governors are designed to operate independently for each system component (e.g., CPU, GPU, or memory). However, the performance of CNN inference is jointly determined by the operations of the components. Therefore without the coor-

dination between governors of different components, it is hard to achieve near-optimal energy efficiency on mobile devices.

To address these challenges, we conduct an extensive measurement study on energy efficiency when running various CNN models on a mobile device by adjusting CPU frequency and memory bandwidth together. We analyze the behavior of those CNN models by their computation blocks (i.e., in a block-wise manner) given that modern CNNs are stacked with several block units, each of which has a specific pattern of layers implemented (detailed in §2.2). From our analysis, we present the following key observations:

(i) Each block in the state-of-the-art CNN models reveals a distinct energy-delay[1] trade-off when scaling CPU frequency or memory bandwidth. For instance, as the front and rear blocks of a model are highly memory-intensive, it becomes more energy-efficient to adjust memory bandwidth rather than CPU clock frequency when processing those blocks. On the other hand, the intermediate blocks tend to be relatively CPU-intensive, implying that the CPU frequency scaling to a certain degree can be more useful for energy efficiency. In this regard, such a block-wise control in the CPU scaling and memory bandwidth for CNN inferences can provide more chances for higher energy efficiency.

(ii) The memory access rates (MARs) of computational blocks within a single CNN model are different based on the block configurable parameters and the cache-related specification of the mobile AP (application processor) on mobile devices. In other words, mobile APs typically have a small cache size, strictly limiting their caching capability. Thus, the MAR can be estimated by the characteristics of CNN's computation blocks and the HW specifications of a mobile AP, such as last level cache (LLC) size, memory bandwidth, and CPU clock frequency. We find that each computation block's MAR critically affects the energy-delay trade-off when scaling CPU frequency and memory bandwidth.

Based on our observations, we propose **NeuroValve**, a new mobile energy management framework that realizes the CPU frequency and memory bandwidth control by exploiting the neural network architecture for optimizing the energy efficiency of an on-device CNN inference. Figure 1 illustrates an overview of NeuroValve. We implement NeuroValve on Google Pixel 3 and measure energy consumption and delay with a set of state-of-the-art CNN models. Our evaluation results confirm that NeuroValve can save energy consumption by up to 30.3% compared to Android's default setting and 12.8% compared to the SysScale [21], which is the most recent memory-aware DVFS technique.

In summary, we make the following contributions:

- We perform the first extensive measurement on the energy consumption and delay for CNN models with various block-wise configurations (§III). We make two key observations from the mea-

[1] Delay here denotes the time taken to process the block.

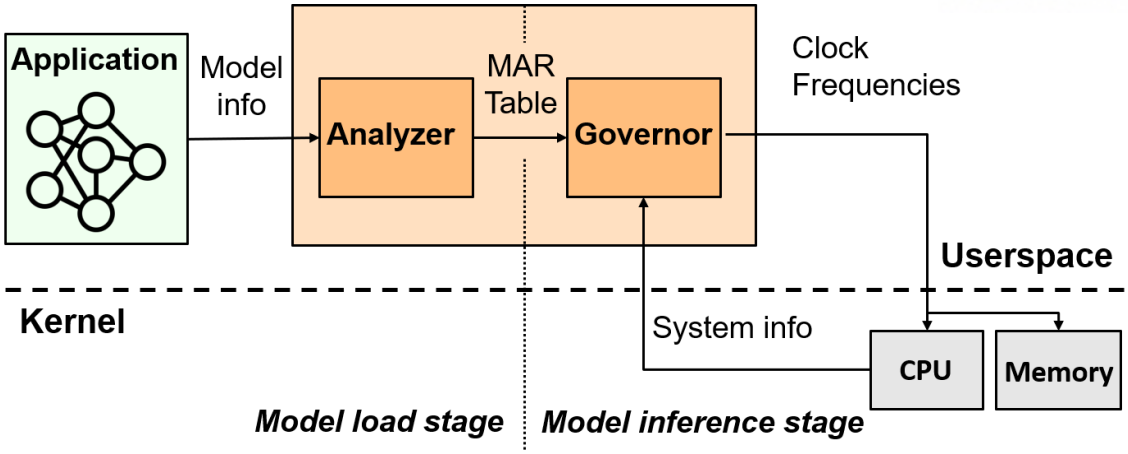


Figure 1: NeuroValve architecture: Analyzer predicts MAR of each block of a given CNN model and Governor performs per-block frequency assignment.

surements: (i) each block in a CNN model shows a distinct energy-delay trade-off when scaling CPU frequency or memory bandwidth. (ii) The MAR from blocks within a CNN model differs based on the block configurable parameters and the mobile AP's cache-related specification.

- We present NeuroValve, a joint CPU and memory frequency scaling framework that improves energy efficiency in CNN inferences by exploiting the neural network architecture (§IV).
- We implement NeuroValve on off-the-shelf smartphones (§V) and conduct extensive evaluations using various state-of-the-art CNN models (§VI). Our evaluation demonstrates that NeuroValve significantly improves energy efficiency with a negligible computation overhead compared to the existing techniques in the Android system.

II Background

In this section, we provide background information on the mobile APs and CNN models designed for mobile devices.

2.1 Mobile AP Architecture

A mobile AP consists of several computation-related components such as multi-core CPU, multi-core GPU, caches therein, and memory controller. The caches are hierarchically designed and include L1 and L2 caches assigned to each CPU core and L3 cache typically shared to all CPU cores. These components are densely packaged together into one small chip due to the limited space for mobile devices. This compact design of mobile APs causes the limit of cache size, leading to frequent access to main memory (i.e., RAM). To mitigate the performance drop caused by frequent external accesses, recent mobile APs are equipped with a system-level cache called last-level cache (LLC) [43, 44, 57, 14] located right next to the system bus. In some cases, L3 cache plays the role of LLC.

LLC is known to improve memory-intensive tasks' performance, including neural network computations, because LLC is located nearer the processor cores than the main memory. Thus it has a faster connection interface to the processors.

However, recent CNN benchmarks on mobile devices show that LLC is far from being sufficient to eliminate the main memory accesses, especially in large CNN models [5]. Interestingly, these main memory accesses caused by cache misses (i.e., LLC misses) affect performance and energy efficiency. This is because scaling up the clock frequencies of CPU cores, which wait for the main memory's slow response, consumes more power. Instead, scaling up the system bus clock frequency will be more beneficial for energy efficiency since the response will arrive quicker. To this end, we raise two important questions: 1) how much the CNN computation is affected by those cache misses and 2) how the energy efficiency of CNN computation is affected by those cache misses.

As we are interested in knowing how frequently the CPU core accesses the memory during processing a CNN workload k, we define the memory access rate from the workload, $MAR(k)$. It is the ratio between the number of LLC misses (N_{llc_misses}) that happened during executing k and the number of floating-point operations (FLOPs) required to execute k as in:

$$MAR(k) = \frac{N_{llc\_misses}(k)}{FLOPs(k)}. \tag{1}$$

Using $MAR(k)$, we empirically study the performance characteristics of running CNN inferences on mobile devices, including the processing delay and the energy consumption, in detail in the next section.

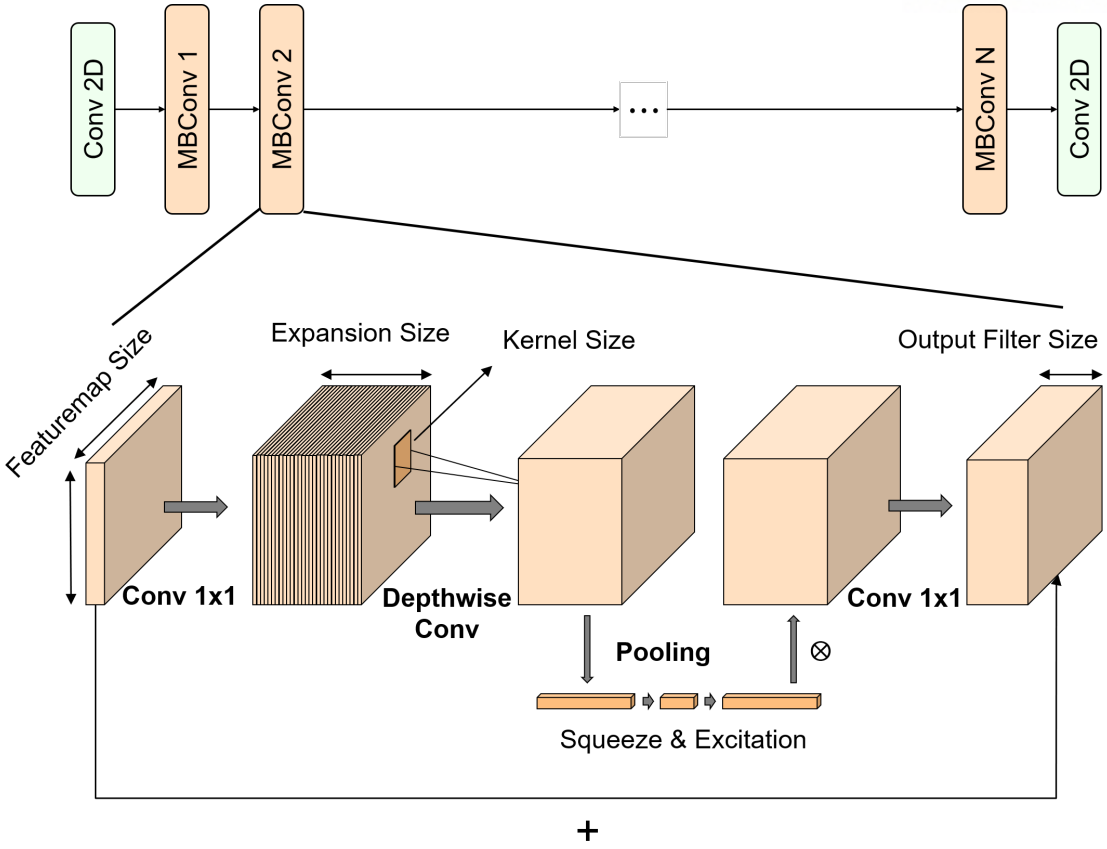


Figure 2: Typical CNN architecture and inverted bottleneck block (MBConv) with its configurable parameters.

2.2 CNN Design leveraging Blocks

Since the advent of CNN models such as ResNet [25], the concept of stacking up a block unit repeatedly on top of each other, has been widely used for design convenience. Therefore, understanding how each block is computed is crucial for understanding how a CNN operates.

Figure 2 depicts a typical mobile CNN architecture and magnifies a block called MBConv (mobile inverted bottleneck convolution) appearing repeatedly. MBConv consists of 1×1 convolutional layer as an input to the block, followed by a depthwise convolutional layer. Then, so-called squeeze & excitation [29] is optionally applied. Here, squeeze & excitation consists of a global average pooling layer and two fully connected layers. After that, the output of the block is finally computed through 1×1 convolutional layer. By doing so, MBConv enlarges the dimension of the input feature space and reduces the dimension again at the end, which leads to a reduction in the memory usage with the help of small-sized input and output tensors. In addition to that, the depthwise convolutional layer therein reduces the number of computations. Thanks to these benefits, the MBConv block is widely adopted in mobile CNNs [48, 27, 55, 56].

However, the block is used in many different ways in practice. There are several parameters to

configure the block to meet the needs of a CNN. These block parameters affect the size of tensors for operating the block. Consequently, it affects FLOPs and the number of memory accesses during its execution. We below explain several key parameters in configuring the MBConv block.

Expansion size is the number of expanded filters from input 1×1 convolutional layer of a block as in Figure 2. As it sequentially affects the depth of the following tensors and the number of weights of all the layers, FLOPs increase proportionally to it. On the other hand, the number of memory access would not linearly increase until the size of tensors no longer fits in the cache. Therefore, if its memory requirements for storing the weights and intermediate results are beyond the caching capability of mobile AP, it would induce a large number of memory accesses.

Feature map size is the width and height of an input tensor. Like the expansion size, FLOPs and the number of memory accesses can increase by enlarging feature map size as it requires large memory space to store input tensors and intermediate tensors for whole layers in a block.

Output filter size is the number of filters of the last 1×1 convolutional layer. FLOPs for the last 1×1 has a large portion of the total FLOPs of a block because the depthwise convolutional layer requires much lower FLOPs for computation than the convolutional layer. However, in terms of memory access, it only affects the size of an output tensor of a block. Thus, increasing output filter size can make large FLOPs than the number of memory accesses.

Kernel size is the width and height of weight tensor for the depthwise convolutional layer. Large kernel size can increase the computation of one convolutional operation. Therefore, increasing the kernel size could result in more increase of FLOPs than the number of memory accesses.

Stride is the step size of sliding the kernel over the feature map during operating of an input 1×1 convolutional layer. Most models use the stride of 1 or 2. If the stride of the depthwise convolutional layer is 1, all of the tensors in a block have the same size in the feature map. However, the stride of 2 halves the output feature map size, and FLOPs also decreases.

Squeeze & excitation is an optional layer that can be included or not. It consists of a global average pooling layer and two fully connected layers. Therefore, if the size of the output tensor of the depthwise convolutional layer is large, the size of tensors for the fully connected layer would be large as well. Thus, the increase in memory access becomes higher than the increase in FLOPs as the fully connected layer needs a large number of weights.

To summarize, the MAR becomes different for each block according to these block parameters. The intricacy between multiple parameters in the CNN block motivates us to deeply understand how the MAR is affected by the block parameter configurations and how it affects the core frequency and memory bandwidth control through extensive measurements.

III Block-wise Energy Efficiency Analysis for CNN Inference

As discussed in §II, CNN block architecture may affect memory-intensity during an inference. To optimize the energy efficiency of CNN inference on mobile devices, we analyze the characteristics of CNN inference at the block-wise level.

3.1 Measurement Setup

To study the impact of model parameters on mobile devices, we conduct measurements using two smartphones, Pixel 3a and Pixel 3, as shown in Table 1. These two devices are chosen because they have different caching capabilities that can present distinct CPU frequency and memory bandwidth control behaviors during a CNN inference. We use the `userspace` governor that allows us to control CPU clock frequency and memory clock frequency in the user space through `sysfs`[2] interface. Note that we control memory bandwidth by controlling the memory clock frequency. For the CPU speed scaling, Pixel 3a and Pixel 3 have 10 and 24 discrete levels. We use all levels for Pixel 3a but use only 12 levels across the whole range for Pixel 3 for simplicity. For the memory bandwidth, Pixel 3a and Pixel 3 have 11 and 7 discrete levels, all of which are used in our measurement.

For mobile CNN execution, we use Tensorflow Lite [15], which is the lightweight deep learning framework used on mobile devices. For accurate energy consumption measurement, we use a Monsoon power monitor [41]. To avoid energy consumption from other components such as screens, network chipsets, and background tasks, we unload all other applications and set the phone to airplane mode. Furthermore, we run CNN inferences with the screen off. We measure the number of last-level cache misses with PMU (performance monitoring unit)[3] physically located inside the AP. For statistical confidence, we repeat the model inference 150 times and present the average as the measurement result. We measure the latency from the `Invoke()` function in Tensorflow Lite, which is called when running inference.

3.2 Energy-Delay Trade-off in CNN's Blocks

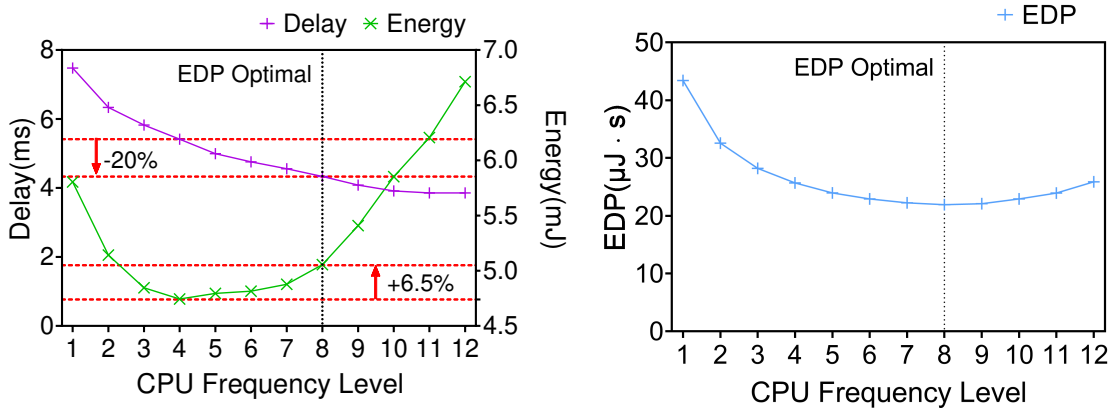
We are interested in observing the impact of CPU frequency scaling and memory bandwidth control with respect to the energy consumption and latency performance in running the CNN inference. For this, we measure the energy consumption and latency of MobileNet V3-Small [27] and EfficientNet-B0 [56],

[2] We use `scaling_setspeed` [6] for CPU frequency setting and `/sys/class/devfreq/soc:qcom,cpubw/userspace/set_speed` for memory clock frequency setting.

[3] We read the PMU register values with `perf_event_open` system call while running inferences.

Device	Pixel 3a	Pixel 3
AP	Snapdragon 670	Snapdragon 845
CPU	Cortex-A55 (1.7GHz) Cortex-A75 (2.0GHz)	Cortex-A55 (1.6GHz) Cortex-A75 (2.5GHz)
Memory	LPDDR4X (Max BW 13.97 GiB/s)	LPDDR4X (Max BW 29.87 GiB/s)
Last-level cache	Shared L3 (1MB)	System Level Cache (3MB)

Table 1: Mobile device specifications



(a) Energy & delay

(b) EDP

Figure 3: Energy consumption, delay, and EDP of Block11 in MobileNet V3-Small on Pixel 3: The lower the better.

which are the state-of-the-art mobile CNN models utilizing MBConv blocks. We analyze their relationship in a block-wise manner. Table 2 shows the detailed configuration of block parameters consisting of the two models.

There are two ways to minimize energy consumption spent to process a task, reducing average power consumption or delay (i.e., time taken to process the task). However, the two metrics are not orthogonal as the efforts to reduce the power consumption (by reducing either the CPU core speed or the memory bandwidth) can lead to an increase in delay. As shown in Figure 3a tested on Pixel 3 for block 11 of MobileNet V3-Small, the delay decreases gradually as the CPU frequency level increases. However, the resultant energy consumption starts increasing again from the 4th CPU frequency level because the power consumption increases more than the reduction in delay. The CPU frequency level that minimizes the energy consumption is somewhere in the middle of its entire range, and this level is far from being the fastest as in Figure 3a. However, energy consumption is not the only thing that users care about. When there are two CPU frequency levels that give similar energy consumption, the delay becomes important as it affects the user's perceived performance.

To this end, we consider both energy and delay performance by adopting a metric called EDP

Block	Parameter
1	(16,112,3,16,2,True)
2	(72,56,3,24,2,False)
3	(88,28,3,24,1,False)
4	(96,28,3,40,2,True)
5	(240,14,5,40,1,True)
6	(240,14,5,40,1,True)
7	(120,14,5,48,1,True)
8	(144,14,5,48,1,True)
9	(288,14,5,96,2,True)
10	(576,7,5,96,1,True)
11	(576,7,5,96,1,True)

(a) MobileNet V3-Small

Block	Parameter
1	(16,112,3,16,1,True)
2	(96,112,3,24,2,True)
3	(144,56,3,24,1,True)
4	(144,56,5,40,2,True)
5	(240,28,5,40,1,True)
6	(240,28,3,80,2,True)
7	(480,14,3,80,1,True)
8	(480,14,3,80,1,True)
9	(480,14,5,112,2,True)
10	(672,7,5,112,1,True)
11	(672,7,5,112,1,True)
12	(672,7,5,192,1,True)
13	(1152,7,5,192,1,True)
14	(1152,7,5,192,1,True)
15	(1152,7,5,192,1,True)
16	(1152,7,3,320,1,True)

(b) EfficientNet-B0

Table 2: Block architecture of MobileNet V3-Small and EfficientNet-B0: The hextuple parameters are expansion size, featuremap size, kernel size, output filter size, stride, and squeeze & excitation from left to right.

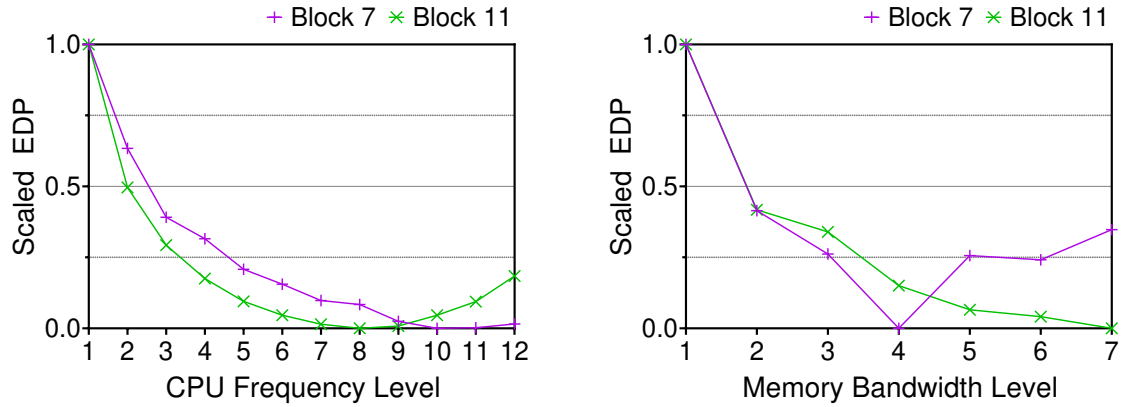
(energy-delay product), the product of energy consumption (E) and delay (D) (i.e., $EDP = E \cdot D$). EDP has been used as an important metric to improve a computing system's energy efficiency while maintaining good performance [51, 52]. Figure 3b shows the CPU frequency level that minimizes EDP is at 8th. At this level, the delay is reduced by almost 20% while the energy consumption increases only 6.5% compared to the level minimizing the energy consumption (i.e., the 4th CPU frequency level).

We now observe how EDP varies depending on the core frequency and memory bandwidth in several blocks widely used in existing CNN models.

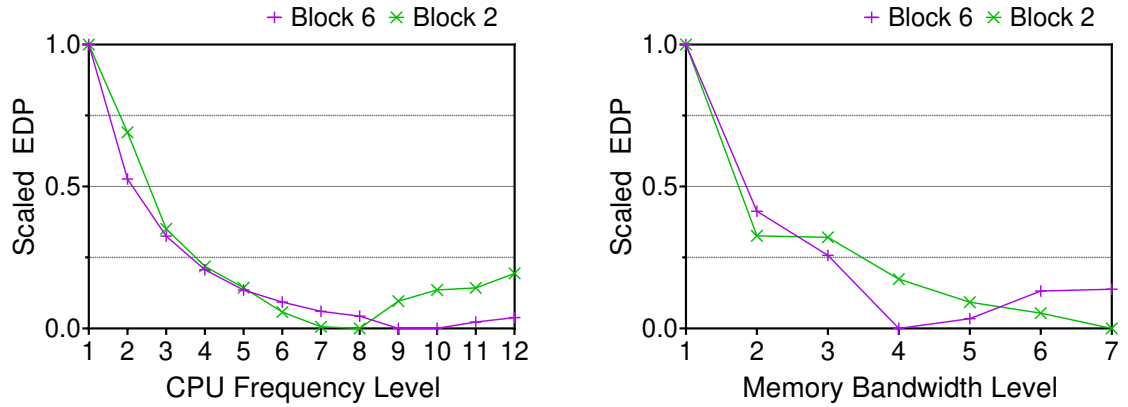
Figure 4 shows the scaled EDP[4] of two blocks in MobileNet V3-Small (Block7 and Block11) and EfficientNet-B0 (Block6 and Block2) by changing the CPU frequency and memory bandwidth levels in Pixel 3a and Pixel 3[5]. We choose those blocks as they reveal distinct trade-off relationships. Specifically, the left figures indicate the scaled EDP with varying CPU frequency levels at the minimum memory bandwidth. In contrast, the right ones show the scaled EDP with varying memory bandwidth at the maximum CPU frequency. The two blocks show different characteristics within the same model as the CPU

[4]We set the minimum and the maximum EDP as 0 and 1, respectively, and map the intermediate values proportionally to [0,1] range.

[5]Block7 and Block6 are intermediate blocks which have relatively small tensors in MobileNet V3-Small and EfficientNet-B0, respectively, but Block2 and Block11 are at the front and the end, which have large tensors.



(a) MobileNet V3-Small

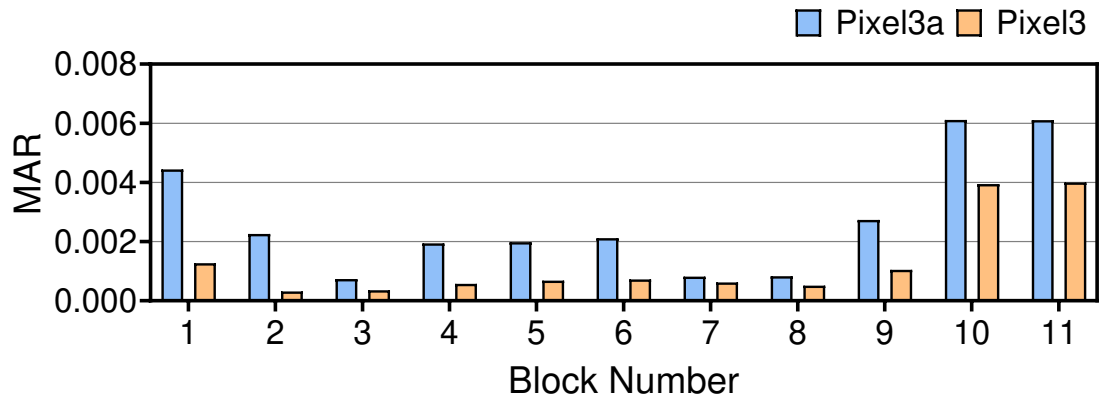


(b) EfficientNet-B0

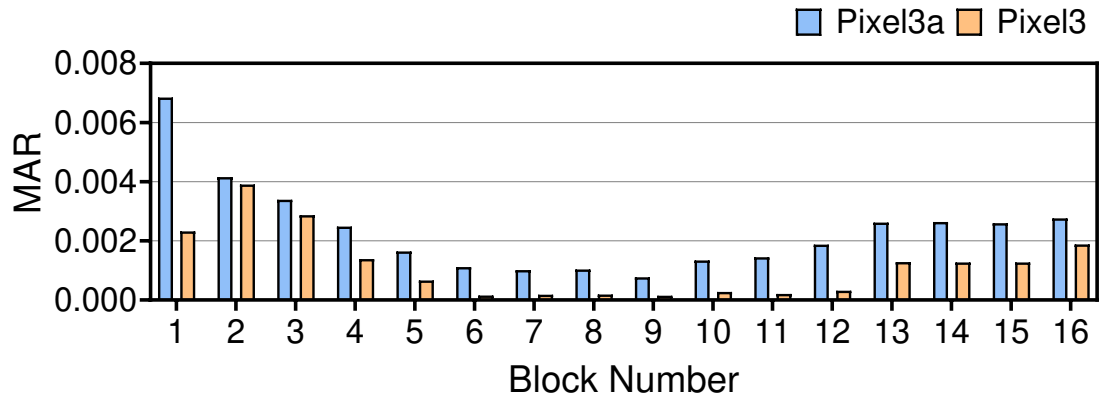
Figure 4: Scaled EDP in MobileNet V3-Small and EfficientNet-B0 w.r.t. CPU frequency level (left) and memory bandwidth level (right) on Pixel 3.

frequency and memory bandwidth change. In Block7 of MobileNet V3-Small, EDP decreases as the CPU frequency increases, while in Block11, EDP increases again after CPU frequency level 8, which is an intermediate level, not the highest. The reason is that Block11's power consumption overwhelms the benefit obtained from increasing the CPU frequency. On the other hand, as the memory bandwidth increases, Block7's EDP increases beyond the memory bandwidth level 4. However, Block11's EDP keeps decreasing despite the increase in memory bandwidth. This implies that processing Block7 becomes less efficient when unnecessarily growing the memory bandwidth. A similar phenomenon is observed in EfficientNet-B0: Block6 of EfficientNet-B0 and Block7 of MobileNet V3-Small have similar patterns, and Block2 and Block11 do as well. We provide more details in §3.3.

In summary, Block7 of MobileNet V3-Small and Block6 of EfficientNet-B0 are more affected by CPU frequency than memory bandwidth in terms of EDP, while Block11 and Block2 in the respective models are more affected by memory bandwidth. This means that Block11 and Block2 are relatively more memory-intensive compared to Block7 and Block6.



(a) MobileNet V3-Small



(b) EfficientNet-B0

Figure 5: MAR in MobileNet V3-Small and EfficientNet-B0 on Pixel 3a and Pixel 3: MAR varies by block and further varies by device for the same block.

Implication. We observe that even within one CNN model, each block has a different energy-delay trade-off depending on the CPU frequency and memory bandwidth. Any single rule on those two scaling factors is unlikely to achieve the optimal EDP because a single rule cannot reflect the distinct trade-offs across different blocks. Therefore, we argue that it is necessary to perform CPU clock frequency and memory bandwidth scaling for CNN inferences in a block-wise manner.

3.3 Memory Access Rate Analysis

As we discussed in §2.2, memory bandwidth can be over-utilized depending on the configurations of a block, which can cause wastes in power consumption from memory and CPU. Thus, it becomes crucial to predict each block's memory bandwidth requirement to reduce excessive resource usages. To estimate the memory bandwidth requirement, we focus on the MAR value as we discussed in §2.1. By varying block parameters in the model, we see how they affect the MAR values of blocks.

3.3.1 MAR for Models

We analyze the MAR value of blocks in MobileNet V3-Small and EfficientNet-B0 (whose architectures are shown in Table 2). The higher the MAR value, the more intensive the memory access required for each FLOP. For high memory access rate, relatively slow memory access time[6] can become a bottleneck, so that increasing the CPU frequency would unnecessarily boost power consumption. On the other hand, in the case of a low memory access rate, excessively increasing the memory bandwidth only loses efficiency.

Figure 5 compares the MAR measurements between two phones for all the blocks of MobileNet V3-Small and EfficientNet-B0, respectively. As shown, they have a wide range of MAR values depending on the block type and mobile device type. Interestingly, both CNN models show quite similar patterns, which have relatively low MAR in the middle of the models and high MAR in the front and the rear of the models. Based on this observation, we find an opportunity to improve energy efficiency by applying different scaling policies across the whole blocks in a CNN model.

Another observation is that MAR values on Pixel 3a are always higher than Pixel 3 in all the two models' blocks. Although FLOPs, when estimated only by the CNN model parameters of a block, would be the same, the actual MAR on Pixel 3a is always higher than Pixel 3. This is because Pixel 3a has a smaller L3 cache size than Pixel 3, as in Table 1. Furthermore, Pixel 3 has a system-level cache (i.e., LLC), and Pixel 3a does not. This makes the larger last level cache misses on Pixel 3a than Pixel 3. This observation implies that it is important to take the mobile AP's specifications into account when predicting a block's memory access rate accurately.

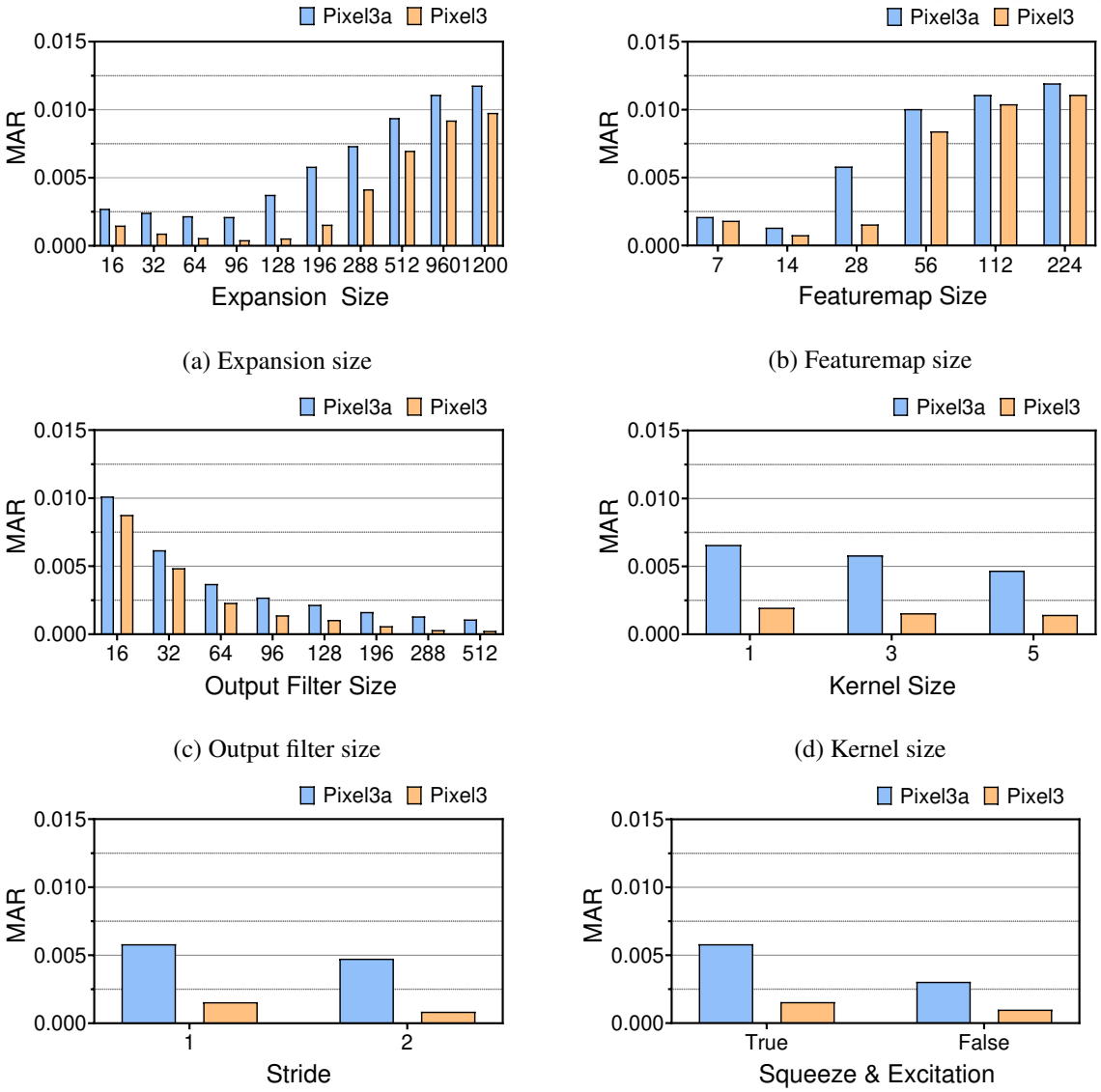
Parameter	Values
Expansion Size	16, 32, 64, 96, 128, 196, 288, 512, 960, 1200
Output Filters	16, 32, 64, 96, 128, 196, 288, 512
Featuremap Size	7, 14, 28, 56, 112, 224
Kernel Size	1, 3, 5
Stride	1, 2
Squeeze & Excitation	True, False

Table 3: CNN's block parameter settings

3.3.2 MAR for Block Parameters

As discussed in §2.2, every block is parameterized. Here, we study the impact of those block parameters on the MAR performance.

[6]We measure the memory latency by using TinyMemBench [49]. The measurement time for random memory access of 16KB that can fit in L1 cache takes 1.1ns on average and for 8MB that cannot fit in the last-level cache takes 110.9ns on average in Pixel 3.



(a) Expansion size

(b) Featuremap size

(c) Output filter size

(d) Kernel size

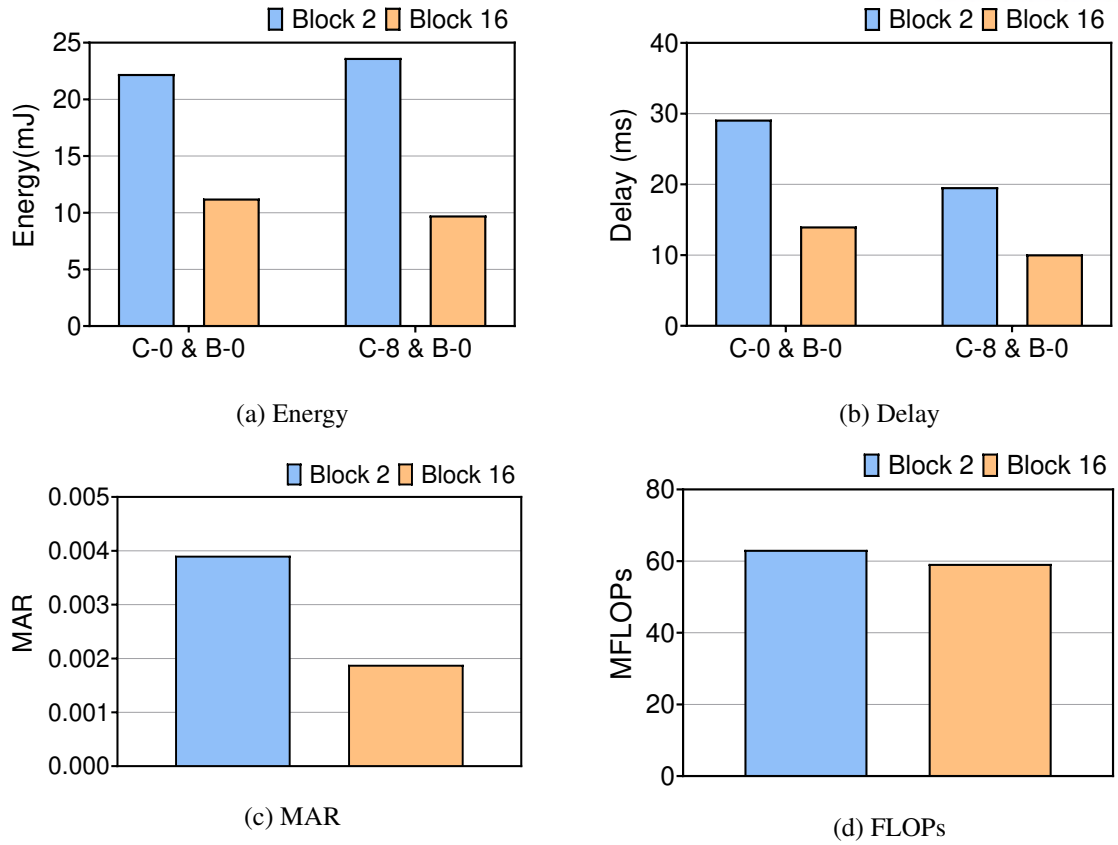
(e) Stride size

(f) Squeeze & Excitation

Figure 6: MAR measurements over different parameters characterizing CNN's block.

We conduct extensive measurements using Pixel 3 and Pixel 3a for all parameters in Table 3, which are generally used by blocks in DNNs, including the state-of-the-art CNN models [27, 56, 55].

Figure 6 compares the MAR measurements between the two phones with respect to various block parameters. As the size of all parameters (except for stride) increases, the computation demand becomes larger, and thus the FLOPs grows. In particular, Figures 6a and 6b show that overall, the MAR values achieved by the two phones increase with the expansion size and the feature map size, respectively. Interestingly, when the expansion and the feature map sizes increase over a small range, MAR is maintained at a low level that can be handled by the last-level cache's capability. Meanwhile, when the parameter exceeds a certain threshold (e.g., in case of the expansion size, 128 for Pixel 3a and 196 for pixel 3), the MAR increases rapidly. The difference in the threshold between the two devices comes from the



(a) Energy

(b) Delay

(c) MAR

(d) FLOPs

Figure 7: Performance comparison between two combinations of CPU frequency and memory bandwidth in Block2 and Block16 of EfficientNet-B0 on Pixel 3: C-*x* & B-*y* means CPU frequency level *x* and memory bandwidth level *y*.

difference in the last-level cache's size.

We further study the MAR values with varying size of kernel, output filter, and stride, respectively, as in Figures 6c, 6d, and 6e. From these, we find that the MAR values reveal a decreasing pattern with the increasing parameters because FLOPs increase much more than the number of memory access as we discussed in §2.2. Figure 6f shows that the MAR value increases with adding squeeze & excitation to a block as it adds a fully connected layer that requires a large number of weights. This result further confirms that the increase of the FLOPs is much larger than the LLC increase for these parameters. Thus, when these parameters become larger, adjusting the CPU clock frequency has a greater impact on energy efficiency than the memory bandwidth.

Implication. We observe that each parameter is highly correlated to the MAR value of the devices. Specifically, the expansion size and feature map size largely affect the MAR of a block. Also, we observe that the MAR value of the same block varies by the difference in memory specification of devices. Thus, we argue that it is necessary to consider the block parameters and the type of mobile AP for an accurate MAR estimation.

3.4 Improving Energy Efficiency with MAR

So far, we have observed that MAR is affected by block configurations and the type of mobile AP. Besides, the blocks which have different MAR values have different energy-delay trade-offs as we observed in §3.2 and §3.3. Therefore, it is highly likely that the predicted MAR values from the blocks of a CNN can provide a hint to control CPU frequency and memory bandwidth in an energy-efficient manner for CNN inferences.

To see the feasibility of improving energy efficiency with respect to MAR, we observe the energy consumption for different combinations of CPU frequency and memory bandwidth in two blocks. Figure 7 shows the energy consumption for two pairs of CPU frequency and memory bandwidth levels in two blocks of EfficientNet-B0 (Block2 and Block16) on Pixel 3. C-*x* & B-*y* denote the combination of CPU frequency level *x* and memory bandwidth level *y*. These blocks are chosen because they have different MAR values with similar FLOPs, which implies that they have a different number of memory accesses during an inference. Interestingly, they show largely different energy consumption at the same CPU frequency and memory bandwidth, even if they have similar FLOPs. Especially for C-0 & B-0, Block2 shows the double for energy consumption. This mainly stems from a long delay caused by low memory bandwidth while experiencing high MAR. We learn from this test that MAR is an essential factor to estimate the energy consumption and delay of a block.

To sum up, considering MAR with CPU frequency and memory bandwidth settings can allow us to predict the energy and delay behavior of a block during an inference. Therefore, if the MAR value of a block can be predicted, it would guide us to choose a better CPU frequency and memory bandwidth combination in energy efficiency.

IV NeuroValve System

4.1 Overview

Motivated by our observations, we develop NeuroValve, a joint CPU clock frequency and memory bandwidth control framework that optimizes the energy efficiency of on-device CNN inferences. The key idea of NeuroValve is to eliminate excessive CPU/memory utilization by predicting the energy and delay consumed in each block of a given CNN model, based on extensive measurements obtained across various CNN block configurations.

Design challenges. When realizing NeuroValve on a mobile device, we face two design challenges. First, it is non-trivial to create a model that predicts CPU clock and memory bandwidth requirements for each block because they can vary widely by various block configurations and different device specifications as discussed in §III. Second, even though the resource requirements have been predicted, the predictions can become inaccurate when running with other tasks sharing those resources.

Our approach. We propose a two-stage approach to address these design challenges, consisting of *model load stage* and *model inference stage*.

Figure 1 shows the proposed system architecture. In the model load stage, a CNN model is loaded, and the *Analyzer* module therein reads and analyzes the whole layers of the model. Then, it creates a MAR table along with the blocks. In the model inference stage, the *Governor* module therein makes predictions for each block's energy consumption and delay performance. The predictions are continuously made using the MAR table created in the model load stage. The actual moving average of the MAR is obtained from the system to take the impact of other tasks into account. Based on such predictions, the *Governor* module adjusts core frequency and memory bandwidth in the system. We explain the details of the two main modules below.

4.2 Analyzer

The analyzer module estimates MAR values based on the configured parameters for each block. The estimation should be accurate and lightweight as the computation involved in the analyzer would translate to the overhead of NeuroValve. Therefore, we opt to build a predictor for the analyzer using a decision tree instead of other advanced machine learning techniques. Our predictor is trained with the dataset collected for the comprehensive measurements in §III. The trained predictor has 20 depth with 5202 leaf nodes with R^2 score of 0.96 for the validation dataset, which shows sufficient accuracy. Note that the predictor makes an output (i.e., MAR values for the blocks in a model) only once when the model is first loaded. Therefore, the analyzer's overhead is negligible given that a CNN inference is substantially heavier than the decision tree computation.

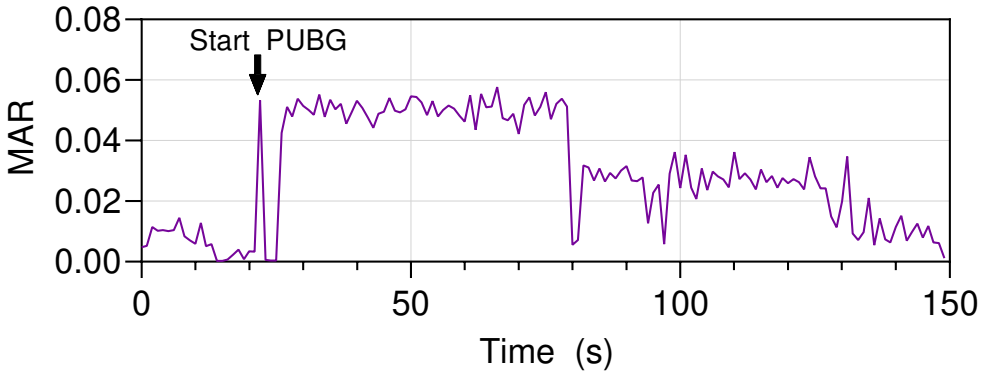


Figure 8: Dynamics of MAR over time with other tasks during the repeated processing of a CNN block: MAR is highly affected by a mobile game (PUBG [35]).

4.3 Governor

As we observed in §3.4, MAR is an essential factor to estimate the energy consumption and delay of a block along with the CPU frequency and memory bandwidth at the moment. Based on the predicted MAR provided by the analyzer module and the actual MAR read from PMU, as a proxy to consider the aggregated impact from processes, our governor module suggests a combination of CPU frequency and memory bandwidth that is expected to minimize EDP.

Figure 8 demonstrates the variation in the actual MAR while repeatedly operating a CNN block with a 3D mobile game launched in the middle. As shown, due to the multiple tasks included in the game, such as rendering and networking functions, the actual MAR fluctuates widely over time. Thus, the governor should consider both MAR values of the predicted one for the CNN block and the actual one for system states in practice to handle the unknown, future workload.

To navigate the tradeoff between delay and power consumption, we introduce the weight $w \in [0,1]$[7] to the original EDP, so that the new metric *wEDP* can traverse via w over two extremes from purely optimizing energy consumption ($w = 1$) to purely optimizing delay ($w = 0$) and the balance in-between, which is defined as:

$$wEDP = E^w \cdot D^{(1-w)}. \tag{2}$$

For an optimal wEDP given both MAR values, the governor module exhaustively estimates the energy consumption and delay of a block for each pair of CPU frequency and memory bandwidth. For the lightweight estimation, we build a dedicated predictor for the governor using another decision tree. It predicts the set of energy consumption and delay of a block given the MAR values for the feasible set of CPU frequency and memory bandwidth. Then, using the set of energy consumption and delay, the governor module finds the pair of CPU frequency and memory bandwidth that achieves *w*EDP-optimality

[7] Note that w can be set by the user or by the application.

given w. The governor finally configures the CPU frequency and memory bandwidth accordingly. The decision tree has 19 depth with 1591 leaf nodes, showing $R^2 = 0.96$ for the validation dataset.

V Implementation

We implement NeuroValve in the user space rather than the kernel space because it is difficult for the kernel to obtain a given CNN model's block configuration parameters without having a customized interface in-between. Specifically, we build our system on Android using Tensorflow Lite [15] framework. Figure 9 depicts the implementation details of our system. Tensorflow Lite makes a `subgraph` from an input model. Then, its `interpreter` object containing the `subgraph` is created. We take the input of our analyzer module from the `interpreter` class so that the analyzer can obtain the block parameters by parsing the tensor information (e.g., the shape of input and output tensors, the pattern of tensors and operation type). Based on such information, our analyzer predicts MAR values for the blocks. In this way, our system profiles the blocks and creates a MAR table in Android once a CNN model is loaded.

When running a CNN inference, `invoke()` function of the `interpreter` is called. Then the `interpreter` traces the `subgraph` from the input node and runs the operation of nodes in order. We take the input of our governor module from the `invoke()` function of the `interpreter`. To consider the impact of other tasks, we let the governor use an additional input (as in §IV) by periodically taking the moving average of the actual MAR values read from the PMU. With those inputs, the governor calculates the *w*EDP-optimal CPU clock frequency and memory bandwidth at the beginning of each block based on its predictions on the energy consumption and the delay.

The governor uses its output to adjust the CPU clock frequency and memory bandwidth through `sysfs` interface. More specifically, it opens the `scaling_set_speed` file for CPU and the `userspace/set_speed` file for memory in the kernel directory and writes the corresponding values for the `userspace` governor to reflect them to the hardware through `cpufreq` and `devfreq` driver.

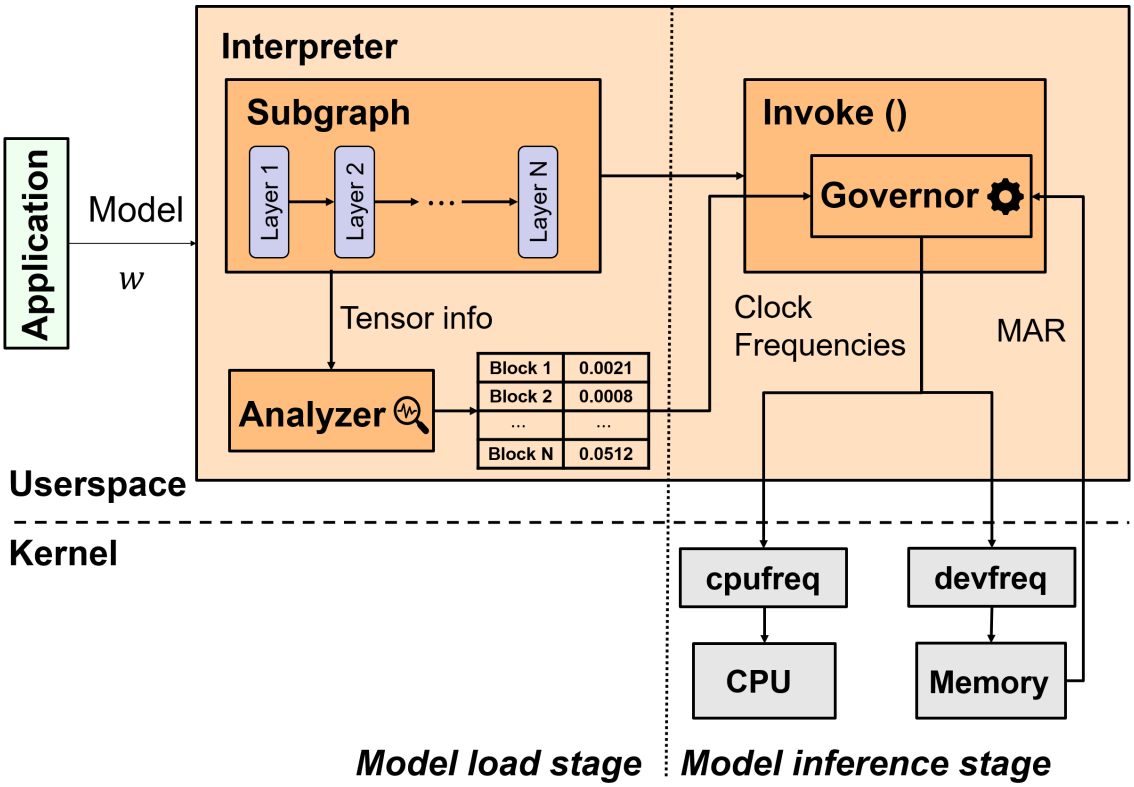


Figure 9: NeuroValve implementation on the Android platform.

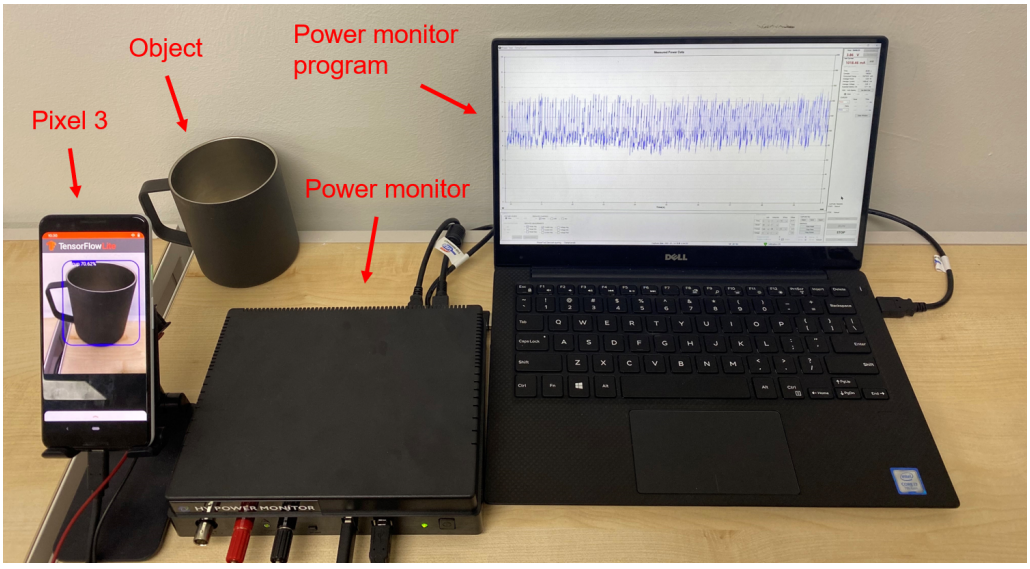


Figure 10: Measurement setup using a Monsoon power monitor and a Pixel 3 smartphone.

VI Evaluation

In this section, we evaluate NeuroValve extensively and verify if it can optimize *w*EDP as described in §IV. We first assess the efficacy of NeuroValve with a CNN inference itself while restricting the impact of background processes. We then evaluate NeuroValve with a mobile application using a CNN in which other resource-consuming tasks such as screen rendering and frame capturing run in the background during the inference.

6.1 Evaluation Setup

We evaluate NeuroValve on Google Pixel 3 (high-end) and Pixel 3a (relatively low-end) with a Monsoon digital power monitor, as shown in Figure 10. Detailed device specification is provided in Table 1. We use several state-of-the-art CNN models for evaluation, including MobileNet V2 [48], MobileNet V3-Small, MobileNet V3-Large [27] and EfficientNet-B0 [56]. To validate that NeuroValve is effective not only for pure CNN inferences but also for a practical application using CNN inferences, we exploit a popular Android object detection application [16] with NeuroValve.

For performance comparison, we use `schedutil` governor [8] for CPU and `bw_hwmon` governor for memory bandwidth which are default governors in recent Android kernels. The `schedutil` increases the CPU frequency to a pre-defined value quickly for fast response and it gradually decreases the frequency when the CPU utilization reduces to a certain threshold. The `bw_hwmon` samples the bandwidth every pre-defined time interval and makes decision based on the sampled bandwidth. We denote them as `default`. In addition, we compare NeuroValve with `SysScale` [21] which is one of the state-of-the-art memory-aware DVFS techniques for mobile devices. The `Sysscale` provides two operation modes (i.e., low

performance and high performance) with pre-defined memory bandwidth based on mobile benchmark workloads. To reduce the excessive resource utilization, it defines the thresholds by adding the average and standard deviation of PMU counter values when the low operation mode's performance degradation goes beyond the thresholds. It periodically samples the PMU counter values, which are averaged over every pre-defined time interval. It also moves from the high-performance mode to the low-performance mode when the averaged PMU value is larger than the thresholds. Then it redistributes the power budget of memory and CPU. By doing so, it improves energy efficiency only with the thresholds. We obtain two thresholds (LLC stalls, LLC misses) for every memory bandwidth level when running CNN inferences for a fair comparison. We set the time interval for sampling PMU values as 30ms, which is a default for `SysScale`.

6.2 Energy Efficiency for CNN Inference

The objective of NeuroValve is to optimize energy efficiency in the perspective of *w*EDP. Experimenting on four CNN models on Pixel 3, Figure 11 shows delay, energy consumption, and EDP (not *w*EDP) of NeuroValve with different w settings (denoted by NV(w)) compared to the Android default governor and `SysScale`. We also present how the usage of CPU frequency and memory bandwidth is distributed while making inferences, as shown in Figure 12. We perform similar experiments using Pixel 3a, and the results are shown in Figure 13.

In Figure 11, NV ($w = 0$), which focuses on minimizing delay, shows that it experiences at least no delay increase or even slight delay reduction for all CNN models compared to the Android default that scales the frequencies to the maximum. In the aspect of energy consumption, NV ($w = 0$) has little benefit compared to other baseline systems because it fully utilizes maximum CPU frequency and memory bandwidth as visualized in Figure 12. Due to the excessive energy consumption to minimize delay, EDP improvement becomes marginal or even lose a little as expected. In contrast, NV ($w = 1$), which mainly uses low-level resources such as minimum memory bandwidth and CPU frequency with its focus on minimizing energy consumption, exhibits a significant gain in energy consumption at the expense of much larger delay, leading to an increase in EDP as shown in Figure 11.

Compared to the aforementioned two extreme cases, NV ($w = 0.5$) pays attention to energy consumption and delay and shows about a 30.26% reduction in energy consumption with only a 9.82% increase in delay for MobileNet V3-Small compared to the Android default and 12.8% compared to `SysScale`. As a result, it achieves a 23.4% improvement in EDP for MobileNet V3-Small and 14.76% improvement in EDP over four models on average. The `SysScale` also improves EDP compared to the Android default by balancing delay and energy consumption to a certain degree. However, its EDP gain is small compared to NV ($w = 0.5$) by a non-trivial margin. It takes a long time for `SysScale` to search

for a good balancing point given a workload, and the workload for CNN inference ((i.e., for blocks being processed in sequence) keeps changing in the millisecond order.

From Figure 12, we find the difference between NV ($w = 0.5$) and `SysScale` in their strategies allocating CPU frequency and memory bandwidth across four CNN models. Regarding the memory bandwidth usage, `SysScale` allocates the maximum bandwidth most of the time. This is because it starts with the maximum bandwidth and keeps adjusting to a lower bandwidth at each interval, which incurs inefficiency in adaptation. So, it allocates much higher memory bandwidth and a lower CPU frequency than NV ($w = 0.5$). On the other hand, NV ($w = 0.5$) can identify which block may suffer from memory bottleneck, thus avoiding allocating excessive memory bandwidth. As a result, NV ($w = 0.5$) achieves greater improvement in EDP with its precise control by understanding workload characteristics from CNN inferences.

Similar observations can be made for Pixel 3a as in Figure 13 showing the performance comparison with two CNN models over the entire range of w.

6.3 Energy Efficiency for Deep Learning Application

We evaluate the efficacy of NeuroValve on a deep learning application for object detection. We build an Android application with TensorFlow Lite for this purpose [16] on Pixel 3, which processes CNN inferences using SSD-MobileNet V3-Small [39, 17] to detect and classify objects from the camera input. With the application, we measure the average delay and power consumption for processing 1000 camera frames. We present the results with the EDP improvement in Figure 14.

Even with tasks other than the CNN inference, Figure 14 shows that NV ($w = 0$) successfully reduces the delay the most while NV ($w = 1$) reduces the power consumption the most. In addition, NV ($w = 0.5$) improves EDP the most as expected, and its improvement far exceeds that from `SysScale`. This is because NeuroValve can consider the memory access bottleneck from the CNN inference and other workloads by referring to the actual MAR as well.

6.4 System Overhead

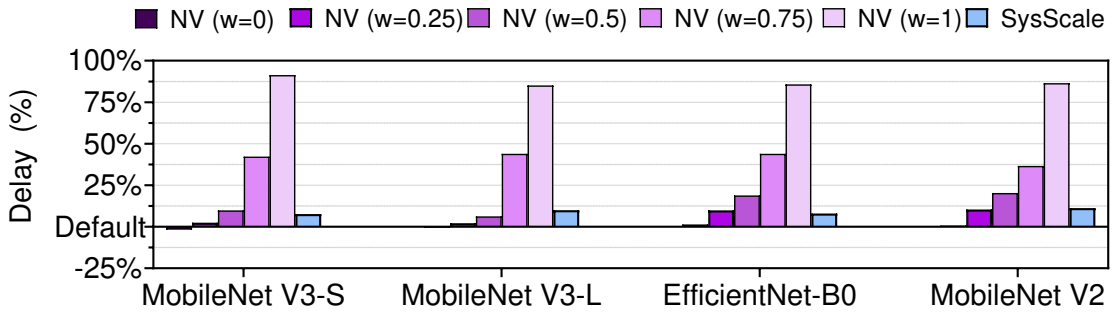
The existing CNN models are known to perform inference in tens of milliseconds on recent off-the-shelf smartphones [31]. Therefore, to apply NeuroValve to mobile devices in practice, the delay overhead from NeuroValve should be reasonably small. Table 4 summarizes the average time to perform an inference on the MobileNet V3-Large model with NeuroValve. When the model is loaded, NeuroValve runs its analyzer and the governor and runs `sysfs` I/O function to adjust CPU frequency and memory bandwidth for each block. We measure the time from the model load to the actual adjustment and consider it as

Function	Time	Portion
Analyzer	0.01ms	0.02%
Governor	0.3ms	0.56%
`sysfs` I/O	0.99ms	1.56%
Inference	62.25ms	97.86%
Total	63.61	100%

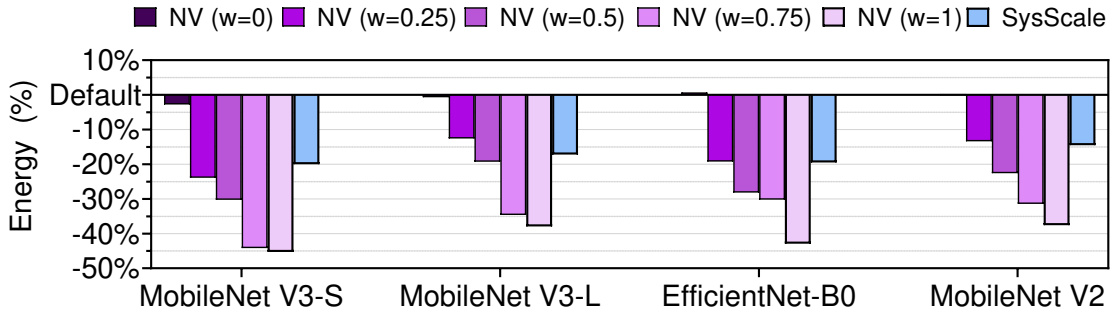
Table 4: System overhead of NeuroValve in time.

overhead. However, note that NeuroValve does not delay any inference because the computation for NeuroValve is made parallel to the inference. This overhead only captures the time till the optimal adjustment is made for each block. As shown in Table 4, thanks to the light computation from our decision tree-based design, the total time overhead of NeuroValve is about 1.3 ms, which accounts for only 2.24% of the total time for a CNN inference[8]. This is acceptable given the improvement in EDP that NeuroValve provides. Moreover, considering the `sysfs` I/O takes the significant portion, we can further reduce the overhead by avoiding to adjust CPU clock frequency and memory bandwidth when the adjustment needs to be made to a neighboring level because such a small change would result in similar energy consumption and delay.

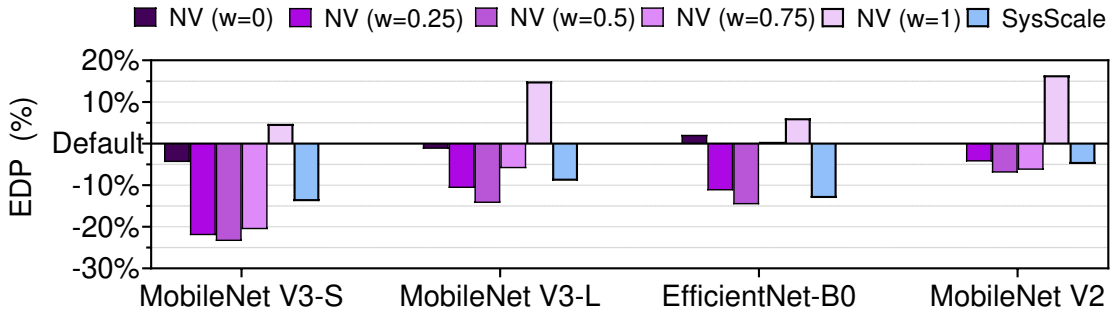
[8]It is worth noting that NeuroValve itself incurs much less overhead than `sysfs` I/O which is beyond our control.



(a) Delay (the lower the better)

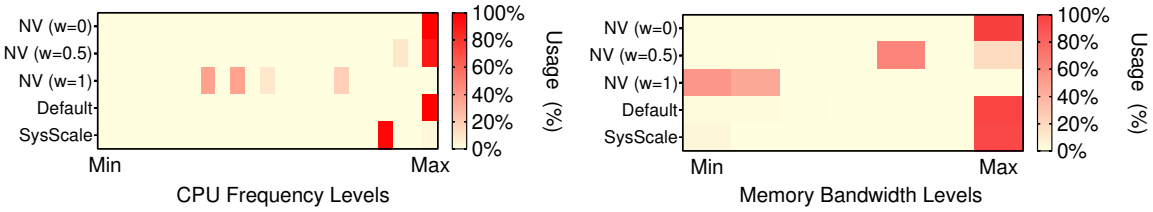


(b) Energy consumption (the lower the better)

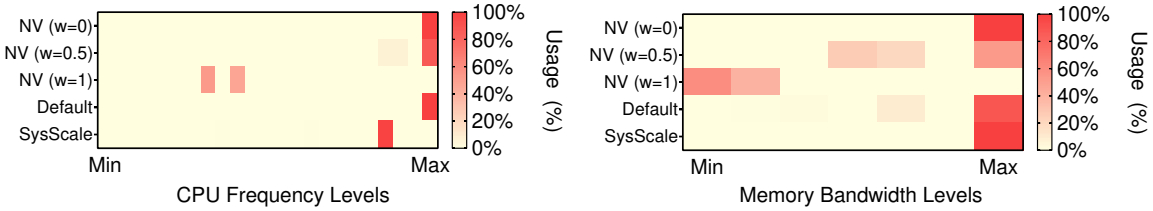


(c) EDP (not *w*EDP for comparison) (the lower the better)

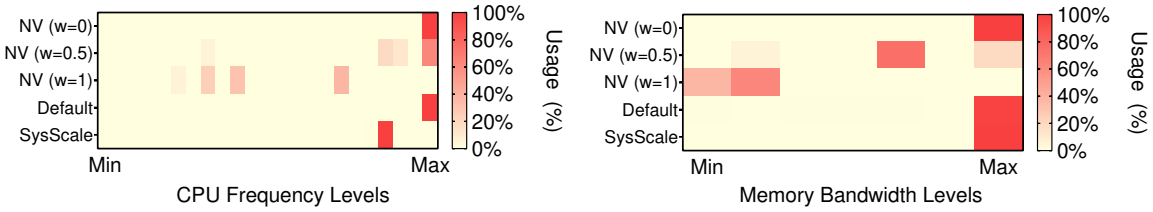
Figure 11: Delay, energy consumption, and EDP comparison of NeuroValve (NV(w)) and SysScale over the Android default in four CNN models on Pixel 3.



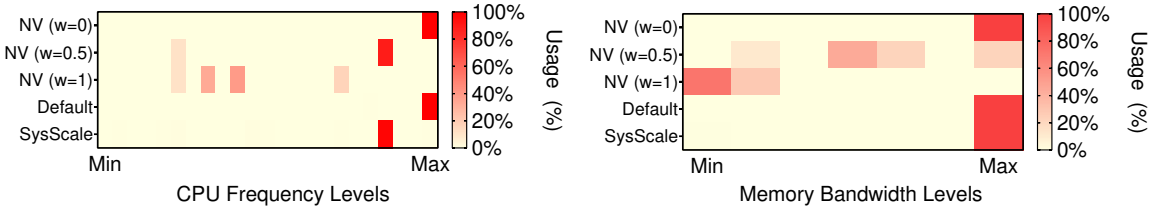
(a) MobileNet V3-Small



(b) MobileNet V3-Large

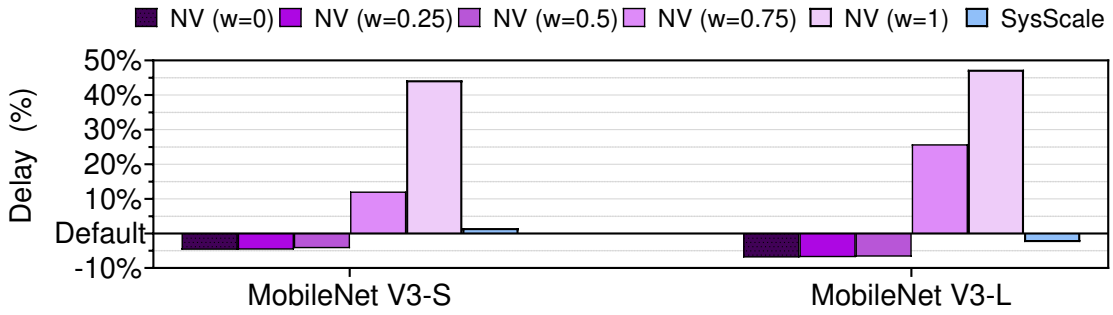


(c) EfficientNet-B0

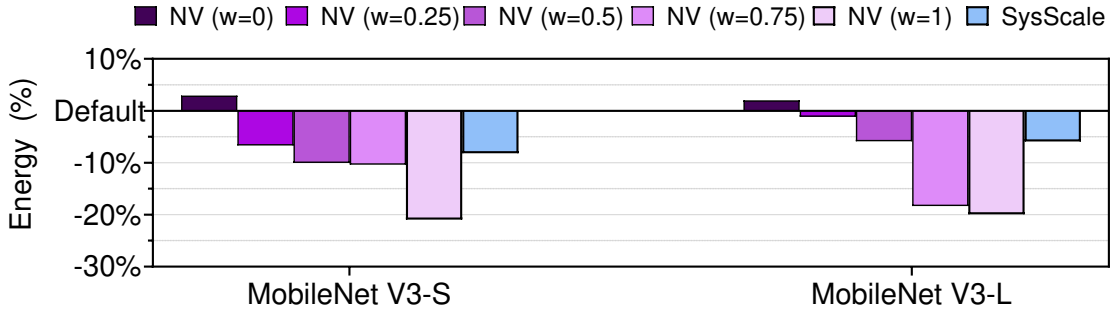


(d) MobileNet V2

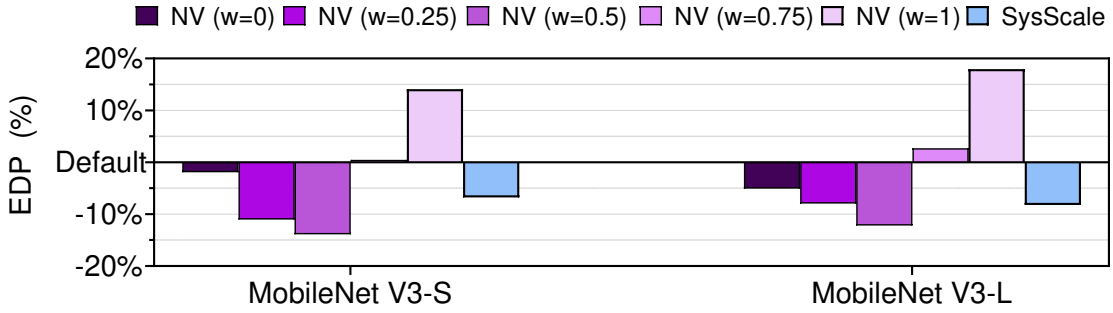
Figure 12: Resource usage distribution of NeuroValve (NV(w)) and other governors during inferences with four CNN models on Pixel 3: CPU frequency (left) and memory bandwidth (right).



(a) Delay (the lower the better)

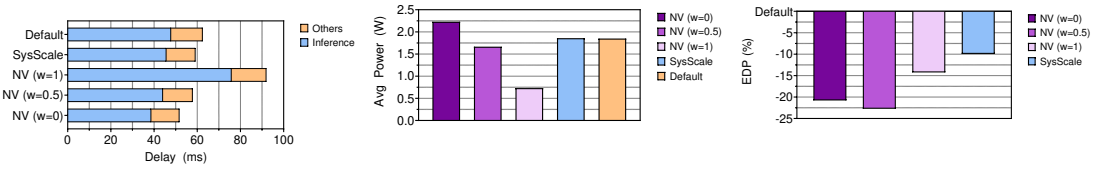


(b) Energy consumption (the lower the better)



(c) EDP (not *w*EDP for comparison) (the lower the better)

Figure 13: Delay, energy consumption, and EDP comparison of NeuroValve (NV(w)) and SysScale over the Android default in two CNN models on Pixel 3a.



(a) Average delay

(b) Average power consumption

(c) EDP improvement

Figure 14: Average delay and power consumption per frame in the object detection application and EDP improvement from NeuroValve and `SysScale` compared to the Android default.

VII Discussion

CNN inference on mobile GPU. Mobile GPU still has limited capacity in its cache memory (e.g., 1MB for Adreno 630 GPU). Also, running inference on mobile GPU requires data memory copy from CPU to GPU, possibly leading to under-utilization of mobile GPU. As a result, main memory access can be the bottleneck for running inferences on mobile GPU. Since the existing governor for GPU clock frequency scaling is independent of memory access [20], understanding MAR can improve energy efficiency in running CNN over mobile GPU. In this sense, our CNN architecture-aware scaling approach would be beneficial to mobile GPU as well.

Multi-DNN. There may exist a particular need to run multiple DNN models simultaneously in an application. For instance, self-driving should continuously detect obstacles while deciding the direction or speed of the car. In this circumstance, focusing only on one model can degrade the overall energy efficiency. For instance, if one model runs a computation-intensive block and another model tries to run a memory-intensive block, it is challenging to decide. This difficulty can be alleviated when adopting the idea of per-core assignment of each DNN, in which NeuroValve is applied to each core differently. How to jointly control them is left for future research.

Extension to other DNN models. We focus our analysis on CNN models. Another prevalent task carried out on mobile devices is sequential processing with recurrent neural network (RNN) models such as speech recognition [18, 19] and activity recognition from sensor data [59, 50]. Such recurrent models also have specific parameters that can affect computation and memory access costs. Therefore, we expect their configurations to exhibit a unique energy-delay trade-off in the frequency scaling, which is also left for future research.

VIII Related Work

Energy efficiency in mobile CPU. Recent research works [11, 38] found that the memory-intensive tasks and CPU-intensive tasks show different energy-performance trade-off relations. They also found metrics determining the memory-intensity of a task and proposed to utilize the metrics to improve energy efficiency. Su et al. [52] proposed a framework that predicts power, performance, and energy across all DVFS states by building a performance model and a per-core power model. Rao et al. [45] observed the energy inefficiency of existing general-pur

pose governors in various Android applications, and built an online controller to run applications energy-efficiently while meeting user-specified performance requirement. However, none of these works has studied or analyzed the processing of neural networks.

Coordinated DVFS. Using DVFS to optimize memory access for energy efficiency has been explored in [13] where the authors studied dynamic scaling of frequencies and voltages of the memory controller, memory channels, and DRAM devices. Deng et al. [12] made a similar proposal by pointing out the necessity to coordinate DVFS controllers of CPU and memory system. Extending both works, SysScale [21] optimizes multiple system components simultaneously by using DVFS to balance and relocate the preserved power according to each subsystem's performance demands.

DVFS in deep learning. Deep learning workloads pose novel optimization challenges coming from their unique computational characteristics [1, 10]. Among others, recent works [3, 4] have investigated the use of DVFS on deep learning systems. PredJoule [4] explored specific performance and energy characteristics of different layers in a DNN and identified the best DVFS configuration for each layer using an embedded platform. NeuOS [3] attempted to simultaneously optimize multiple DNN workloads by balancing energy at the system level via DVFS and accuracy at the application level via configuration adjustments of DNNs. However, to the best of our knowledge, NeuroValve is the first to enable frequency scaling directly from analyzing a deep neural network's block configuration parameters.

IX Conclusion

In this work, we present NeuroValve, a novel CPU frequency and memory bandwidth control framework that understands CNN blocks and exploits this understanding to improve the energy efficiency of CNN inferences. We investigated the impact of various CNN block configurations on energy consumption and delay performance. We found that adaptive CPU frequency scaling and memory bandwidth control per CNN block can significantly improve energy efficiency. We evaluated the efficacy of Neurovalve with various state-of-the-art CNN models and a deep learning application via implementation on the off-the-shelf smartphones. Our evaluations confirmed that NeuroValve could save energy consumption by up to 30.3% compared to the Android default setting and up to 12.8% compared an advanced memory-aware DVFS technique, `SysScale`.

References

[1] ADOLF, R., RAMA, S., REAGEN, B., WEI, G., AND BROOKS, D. Fathom: reference workloads for modern deep learning methods. In *Proceedings of 2016 IEEE International Symposium on Workload Characterization (IISWC)* (2016), pp. 1–10.

[2] BADRINARAYANAN, V., KENDALL, A., AND CIPOLLA, R. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence* (2017).

[3] BATENI, S., AND LIU, C. NeuOS: A Latency-Predictable Multi-Dimensional Optimization Framework for DNN-driven Autonomous Systems. In *Proceedings of 2020 USENIX Annual Technical Conference (ATC 20)* (2020), pp. 371–385.

[4] BATENI, S., ZHOU, H., ZHU, Y., AND LIU, C. Predjoule: A timing-predictable energy optimization framework for deep neural networks. In *Proceedings of 2018 IEEE Real-Time Systems Symposium (RTSS)* (2018), IEEE, pp. 107–118.

[5] BIANCO, S., CADENE, R., CELONA, L., AND NAPOLETANO, P. Benchmark analysis of representative deep neural network architectures. *IEEE Access 6* (2018), 64270–64277.

[6] BRODOWSKI, D. Linux CPUFreq User Guide. `https://www.kernel.org/doc/Documentation/cpu-freq/user-guide.txt`.

[7] BRODOWSKI, D., GOLDE, N., WYSOCKI, R. J., AND KUMAR, V. Linux CPUFreq Governors. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`.

[8] BROWN, N. Improvements in CPU frequency management. `https://lwn.net/Articles/682391/`.

[9] CHEN, L.-C., PAPANDREOU, G., KOKKINOS, I., MURPHY, K., AND YUILLE, A. L. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence* (2017).

[10] CHEN, T., MOREAU, T., JIANG, Z., ZHENG, L., YAN, E., SHEN, H., COWAN, M., WANG, L., HU, Y., CEZE, L., GUESTRIN, C., AND KRISHNAMURTHY, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 578–594.

[11] CHOI, K., SOMA, R., AND PEDRAM, M. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of International symposium on Low power electronics and design* (2004).

[12] DENG, Q., MEISNER, D., BHATTACHARJEE, A., WENISCH, T. F., AND BIANCHINI, R. Coscale: Coordinating cpu and memory system dvfs in server systems. In *2012 45th annual IEEE/ACM international symposium on microarchitecture* (2012), IEEE, pp. 143–154.

[13] DENG, Q., MEISNER, D., RAMOS, L., WENISCH, T. F., AND BIANCHINI, R. Memscale: active low-power modes for main memory. *ACM SIGPLAN Notices 46*, 3 (2011), 225–238.

[14] FRUMUSANU, A. The Apple iPhone 11, 11 Pro 11 Pro Max Review: Performance, Battery, Camera Elevated. `https://www.anandtech.com/show/14892/the-apple-iphone-11-pro-and-max-review/3`, 2019.

[15] GOOGLE. Tensorflow Lite. `https://www.tensorflow.org/lite/`, 2019.

[16] GOOGLE. Tensorflow Lite object detection. `https://www.tensorflow.org/lite/examples`, 2019.

[17] GOOGLE. TensorFlow 1 Detection Model Zoo. `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md`, 2020.

[18] GRAVES, A., AND JAITLY, N. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of International conference on machine learning (ICML)* (2014).

[19] GRAVES, A., MOHAMED, A.-R., AND HINTON, G. Speech recognition with deep recurrent neural networks. In *Proceedings of 2013 IEEE international conference on acoustics, speech and signal processing* (2013).

[20] GUPTA, A. All about CPU Governors and GPU governors. `http://ajgupta.github.io/android/2015/01/28/CPU-and-GPU-governors/`.

[21] HAJ-YAHYA, J., ALSER, M., KIM, J., YAGLIKÇI, A. G., VIJAYKUMAR, N., ROTEM, E., AND MUTLU, O. SysScale: Exploiting Multi-domain Dynamic Voltage and Frequency Scaling for Energy Efficient Mobile Processors. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020).

[22] HAN, S., LIU, X., MAO, H., PU, J., PEDRAM, A., HOROWITZ, M. A., AND DALLY, W. J. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News 44*, 3 (2016), 243–254.

[23] HAN, S., MAO, H., AND DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[24] HE, K., GKIOXARI, G., DOLLÁR, P., AND GIRSHICK, R. Mask r-cnn. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)* (2017).

[25] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[26] HE, Y., ZHANG, X., AND SUN, J. Channel pruning for accelerating very deep neural networks. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)* (2017).

[27] HOWARD, A., SANDLER, M., CHU, G., CHEN, L.-C., CHEN, B., TAN, M., WANG, W., ZHU, Y., PANG, R., VASUDEVAN, V., LE, Q. V., AND ADAM, H. Searching for mobilenetv3. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)* (2019).

[28] HOWARD, A. G., ZHU, M., CHEN, B., KALENICHENKO, D., WANG, W., WEYAND, T., ANDREETTO, M., AND ADAM, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[29] HU, J., SHEN, L., AND SUN, G. Squeeze-and-excitation networks. In *Proceedings of IEEE conference on computer vision and pattern recognition* (2018), pp. 7132–7141.

[30] IANDOLA, F. N., HAN, S., MOSKEWICZ, M. W., ASHRAF, K., DALLY, W. J., AND KEUTZER, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360* (2016).

[31] IGNATOV, A., TIMOFTE, R., CHOU, W., WANG, K., WU, M., HARTLEY, T., AND VAN GOOL, L. Ai benchmark: Running deep neural networks on android smartphones. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops* (September 2018).

[32] JACOB, B., KLIGYS, S., CHEN, B., ZHU, M., TANG, M., HOWARD, A., ADAM, H., AND KALENICHENKO, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2018).

[33] JOUPPI, N. P., YOUNG, C., PATIL, N., PATTERSON, D., AGRAWAL, G., BAJWA, R., BATES, S., BHATIA, S., BODEN, N., BORCHERS, A., ET AL. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), pp. 1–12.

[34] KANG, Y., HAUSWALD, J., GAO, C., ROVINSKI, A., MUDGE, T., MARS, J., AND TANG, L. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[35] KRAFTON. PUBG MOBILE. `https://www.pubgmobile.com/en-US/`.

[36] LASKARIDIS, S., VENIERIS, S. I., ALMEIDA, M., LEONTIADIS, I., AND LANE, N. D. Spinn: Synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (2020).

[37] LI, H., KADAV, A., DURDANOVIC, I., SAMET, H., AND GRAF, H. P. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).

[38] LIANG, W.-Y., AND LAI, P.-T. Design and implementation of a critical speed-based dvfs mechanism for the android operating system. In *Proceedings of 5th International Conference on Embedded and Multimedia Computing* (2010).

[39] LIU, W., ANGUELOV, D., ERHAN, D., SZEGEDY, C., REED, S., FU, C.-Y., AND BERG, A. C. Ssd: Single shot multibox detector. In *Proceedings of European conference on computer vision* (2016), Springer, pp. 21–37.

[40] LONG, J., SHELHAMER, E., AND DARRELL, T. Fully convolutional networks for semantic segmentation. In *Proceedings of IEEE conference on computer vision and pattern recognition (CVPR)* (2015).

[41] MONSOON-SOLUTIONS. High Voltage Power Monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`, 2019.

[42] PARASHAR, A., RHU, M., MUKKARA, A., PUGLIELLI, A., VENKATESAN, R., KHAILANY, B., EMER, J., KECKLER, S. W., AND DALLY, W. J. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News 45*, 2 (2017), 27–40.

[43] QUALCOMM. Snapdragon 845 Mobile Platform. `https://www.qualcomm.com/products/snapdragon-845-mobile-platform`, 2020.

[44] QUALCOMM. Snapdragon 855 Mobile Platform. `https://www.qualcomm.com/products/snapdragon-855-mobile-platform`, 2020.

[45] RAO, K., WANG, J., YALAMANCHILI, S., WARDI, Y., AND HANDONG, Y. Application-specific performance-aware energy optimization on android mobile devices. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017).

[46] REDMON, J., DIVVALA, S., GIRSHICK, R., AND FARHADI, A. You only look once: Unified, real-time object detection. In *Proceedings of IEEE conference on computer vision and pattern recognition (CVPR)* (2016).

[47] REN, S., HE, K., GIRSHICK, R., AND SUN, J. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Proceedings of Advances in neural information processing systems* (2015).

[48] SANDLER, M., HOWARD, A., ZHU, M., ZHMOGINOV, A., AND CHEN, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of IEEE conference on computer vision and pattern recognition (CVPR)* (2018).

[49] SIAMASHKA, S. TinyMemBench. `https://github.com/ssvb/tinymembench`.

[50] SINGH, D., MERDIVAN, E., PSYCHOULA, I., KROPF, J., HANKE, S., GEIST, M., AND HOLZINGER, A. Human activity recognition using recurrent neural networks. In *Proceedings of International Cross-Domain Conference for Machine Learning and Knowledge Extraction* (2017).

[51] SPILIOPOULOS, V., KAXIRAS, S., AND KERAMIDAS, G. Green governors: A framework for continuously adaptive dvfs. In *Proceedings of International Green Computing Conference and Workshops* (2011), IEEE, pp. 1–8.

[52] SU, B., GU, J., SHEN, L., HUANG, W., GREATHOUSE, J. L., AND WANG, Z. PPEP: Online performance, power, and energy prediction framework and DVFS space exploration. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), pp. 445–457.

[53] SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., AND RABINOVICH, A. Going deeper with convolutions. In *Proceedings of IEEE conference on computer vision and pattern recognition (CVPR)* (2015).

[54] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. In *Proceedings of IEEE conference on computer vision and pattern recognition (CVPR)* (2016).

[55] TAN, M., CHEN, B., PANG, R., VASUDEVAN, V., SANDLER, M., HOWARD, A., AND LE, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2019).

[56] TAN, M., AND LE, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946* (2019).

[57] WIKIPEDIA. Apple A13. `https://en.wikipedia.org/wiki/Apple_A13`, 2021.

[58] XIE, R., JIA, X., WANG, L., AND WU, K. Energy efficiency enhancement for cnn-based deep mobile sensing. *IEEE Wireless Communications 26*, 3 (2019), 161–167.

[59] YAO, S., HU, S., ZHAO, Y., ZHANG, A., AND ABDELZAHER, T. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web* (2017).

[60] YAO, S., LI, J., LIU, D., WANG, T., LIU, S., SHAO, H., AND ABDELZAHER, T. Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems* (2020), p. 476–488.

[61] ZHANG, X., ZHOU, X., LIN, M., AND SUN, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of IEEE conference on computer vision and pattern recognition (CVPR)* (2018).